Development of the Infrared Scene Projection System (IRSP) Digital Model (IDM) Utilizing Object Oriented Methodologies (OOM)

Lisa D. Grelier[1], John N. Montgomery[1], and Kenneth G. LeSueur[2]

[1] 555 Sparkman Drive, Executive Plaza, Suite 1622
Huntsville, AL 35816
{lgrelier, jmontgomery}@rttc.army.mil
[2] US Army, Developmental Test Command
Redstone Technical Test Center
CSTE-DTC-RT-E-SA, Bldg. 4500
Redstone Arsenal, AL 35898-8052
klesueur@rttc.army.mil

Abstract. This paper describes the software development of an Infrared Scene Projector Digital Model (IDM). The IDM is a product being developed for the Common High Performance Computing Software Support Initiative (CHSSI) program under the Integrated Modeling and Test Environments (IMT) Computational Technology Area (CTA). The primary purpose of the IDM is to provide a scalable software model of an Infrared Scene Projector (IRSP). The IDM will be utilized for the development and testing of non-uniformity correction algorithms. The major topics to be discussed will include the Carnegie Mellon University Software Engineering Institute (SEI) Capability Maturity Model (CMM) Level II Procedures and the use of Object Oriented Methodologies (OOM) for the design and implementation of the IDM.

1.0 Introduction

Performance assessment of IR imaging systems has traditionally been accomplished using a combination of laboratory and field testing. Laboratory testing allows the projection of complex and dynamic infrared scenes directly into the aperture of the sensor system in the laboratory, but also provides strict repeatability of such scenes, something impossible to accomplish in the field setting. This provides IR sensor hardware and software developers the ability to iterate their designs. One of the greatest advantages of laboratory testing is the cost savings [4,p.1]. The Infrared Scene Projector (IRSP) Digital Model (IDM) is a software product being developed to simulate primary hardware components of a typical IRSP system. The IDM will be utilized for the development and testing of non-uniformity correction algorithms in the laboratory. Non-Uniformity is defined as noise, blur, or other unwanted artifacts in the projected scene. The IDM was developed using standard Object Oriented Methodologies (OOM) while following the Carnegie Mellon University Software Engineering Institute (SEI) Capability Maturity Model (CMM) Level II Procedures. This paper will describe the main aspects of the CMM. Next, we will discuss the basic concepts of an Object Oriented development. Finally, we will show how these procedures were applied to the development of the IDM from initial concept through testing.

1.1 General Methodologies

1.1.1 Software Engineering Methodologies: Carnegie Mellon University Software Engineering Institute (SEI) Capability Maturity Model (CMM) Level II Procedures

The CMM is a framework that describes the key elements of an effective software process. These key activities can be applied to any software project, regardless of size or complexity [2,p.26]. “The CMM describes an evolutionary improvement path from an ad hoc, immature process to a mature, disciplined process” [1, p.O-7]. The CMM guidelines and policies provide a common framework in which to judge a software development organization’s group process maturity level.

The CMM is composed of five maturity levels. Each of these maturity levels defines a set of principles to follow. These principles address the potential areas of difficulty within a software project. The IDM software development is operating under Level II practices, which is referred to as repeatable. The Level II principles include the following [1, p. O-19] [2,p.27]:

- Software Configuration Management (SCM)

SCM is the practice of assuring the integrity of the software project throughout the development process. One example of an SCM activity is source code control, where all the software is maintained in a controlled environment or database.

- Software Quality Assurance (SQA)

SQA provides the management team visibility into the development process. This can be done through peer group reviews and by following all the software engineering phases, requirements, design, implementation, and testing.

- Software Subcontract Management (SSM)

This area deals with selecting and managing sub-contractors. It does not apply to the IDM project.

- Software Project Tracking and Oversight

This principle involves communication between the software lead and management about schedule conflicts and discrepancies well before they spell doom for the project. Also, the project lead must maintain accurate and organized financials records.

- Software Project Planning

Software project planning includes creating a map of the objectives of the project, resource issues, risk assessment, and schedule deadlines.

- Requirements Management

This principle involves an understanding between the development team and the customer. There must exist a process by which the customer and the developer understand and agree upon the ultimate objectives of the project. Then these objectives should be documented in a Software Requirements Document (SRD).

1.1.2 Object Oriented Concepts

In an object-oriented world everything is classified as an *object.* Hence, the name *object-oriented programming*. An *object* is just a single instance of a *class,* where a *class* could have multiple instances *(objects*) of itself*.* A *class* can be thought of as a *black-box* where all its data characteristics and functionality are enclosed inside, which is a key object oriented concept referred to as *encapsulation*. Other objects in the system communicate with the black-box by using a controlled interface. This interface allows for communication without needing to know the internals of the black-box. *Encapsulation* provides a means to modify the black-box's data and functionality while maintaining the same interface. All internal changes should be transparent to the users communicating with the black-box [3,p.5][5].

The *class* that other objects are derived from is called the *parent* or *base class.* These *objects* inherit all the attributes, data, and functionality of its *parent class* [2,p.354]. *Inheritance* is the ability of an object to build on the foundation of another object. The principals of *inheritance* provide a framework to create complex classes that are built on simpler classes. The data and functionality is written once in the *parent class* and the *derived classes* inherit these characteristics allowing for reuse of code. This makes for a leaner and cleaner code. Consider the following example.

Let the *parent class* be People with attributes including height, weight, and eye color. The actions that People can perform include walk and talk. Let there be two derived classes called Police and Doctor that will inherit base attributes from the parent class People. These two new classes will inherit all of the People class attributes and functionality so there is no need to redefine the characteristics. Along with the inherited behaviors, we will give the Police and Doctor some of their own specific attributes and actions such as rank and area of specialty, respectively.

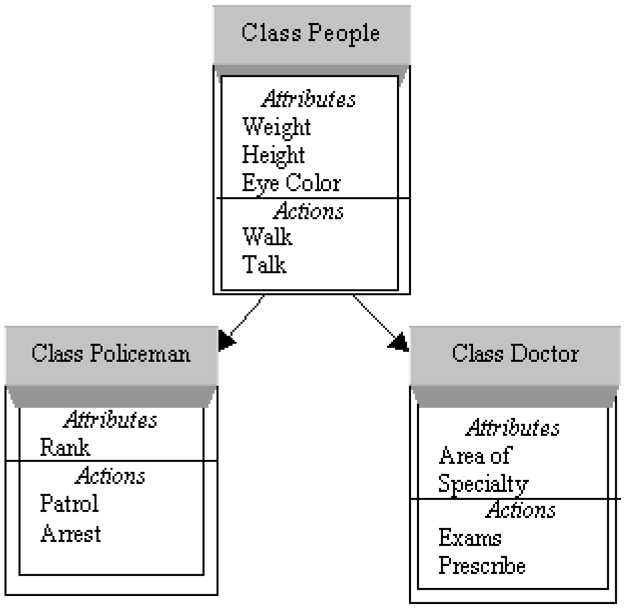

Figure 1 - Class Inheritance

Polymorphism is another object-oriented concept that allows the definition a generic function to be used for multiple purposes. For instance in C++ programming language, a function can have one name but actually do many functions. The data type that is sent to the function determines which particular action the function will perform [3, p.5]. This concept provides the means to handle complex functions with a standard interface. For instance, there may be a function Foo(x: int, real) that takes either an integer or real number as an argument. The action that Foo() will perform depends on which data type is sent.

1.2 The IDM

The requirements and design phases of the software process are probably the most crucial parts of the software development. The requirements and design phases are the foundation of software development. Just like building a house, the foundation must be solid else problems will arise later. From these two phases, follows the implementation and testing, which should go more smoothly with a good foundation. However, note that this can be an iterative process. Once, you get to the implementation phase you might notice the need for another requirement or the need to modify the design. Also, during the testing phase bugs will be found; bugs are what you are looking for anyhow. So the developer must go back at least to the implementation phase if not further to correct the problem.

1.2.1 Requirements

Essentially what should be accomplished in the *requirements* phase, is data gathering and analysis to understand the specific operations the software should perform. This includes required functionality, performance, user and hardware interfacing, and behavior of the system. After the developer has gathered the data, the first item produced is a Software Development Plan (SDP) Document that includes the test procedures, process schedules, and requirements management. The requirements are then documented and given a specific number to be used later in the development [2,p.31]. The list of requirements, an explanation of each, the requirements table, and any necessary technical information for the IDM is written up in a Software Requirements Document (SRD) according to the CMM.

Since, the IDM is the software representation of a typical IRSP hardware system each major component in the IRSP should be represented, this would be considered “required functionality”. Thus, the IDM is made up of the basic hardware components that include the Computer Image Generator (CIG), Control Electronics Subsystem (CES), Infrared Emitter Subsystem (IRES), Projection Optics Subsystem (POS), and Non-Uniformity Correction Subsystem (NUCS) [4,p.1].

Once the requirements are defined, a requirement will be given a specific identifier. This identifier will be used throughout the development process for cross-reference purposes. The requirement description should be as detail as possible. Too vague a requirement may mislead the developer to think a different action is to be performed and thus incorrectly implemented. Definition of any unfamiliar terms is also good practice. It is important to talk to the customer numerous times to clarify the requirements. Also, be sure that the requirement is feasible. The following table displays a list of example requirements and identifiers.

Req. ID	**Requirements**
1.5.2.1	Must represent the basic IRSP hardware components: Resister Array, Camera, Projection Optics, Control Electronics.
1.5.2.2	Scaled speedup exceeds 40% of optimum on 8 processors or greater. Definition: Optimum Theory is defined as 1 x the work can be done on 1 processor then optimally 8X the work should be able to be done on 8 process.
1.5.2.3	Parallel speedup reduces clock time by 2 times on an 8 processor system compared to a single processor system
1.5.2.4	IRSP digital model (IDM) & Scene-based Infrared Scene Generator (IRSG) NUC (SBNUC)Codes will run on two HPC platforms with same valid results
1.5.2.5	All documents and source code under configuration management
1.5.2.6	Option to output derived pixel data in a window

Figure 2 - Requirements Matrix

1.2.2 Design

“The *design* phase focuses on the software data structures, software architecture, interface representation, and algorithm details“ [2,p.31]. The design translates each of the requirements into a representation that can be analyzed before implementation takes place. This process may be iterative. The design is also documented in the Software Design Document (SDD) and becomes part of the software configuration as required by CMM Level II [2,p.31].

Rational Rose, developed by Rational Software Corporation*, a* software product based on the Unified Modeling Language (UML) was used in the design phase of the IDM. This software package allows the user to develop class-diagrams, flow charts, and use-case diagrams for interfaces. The above requirement 1.5.2.1 lists the necessary hardware components needed to accurately represent the IDM. Following the OOM, each one of these hardware components was represented as a class in the class diagram, Figure 3. In addition, the flow chart, Figure 4, shows how the components interact with each other.

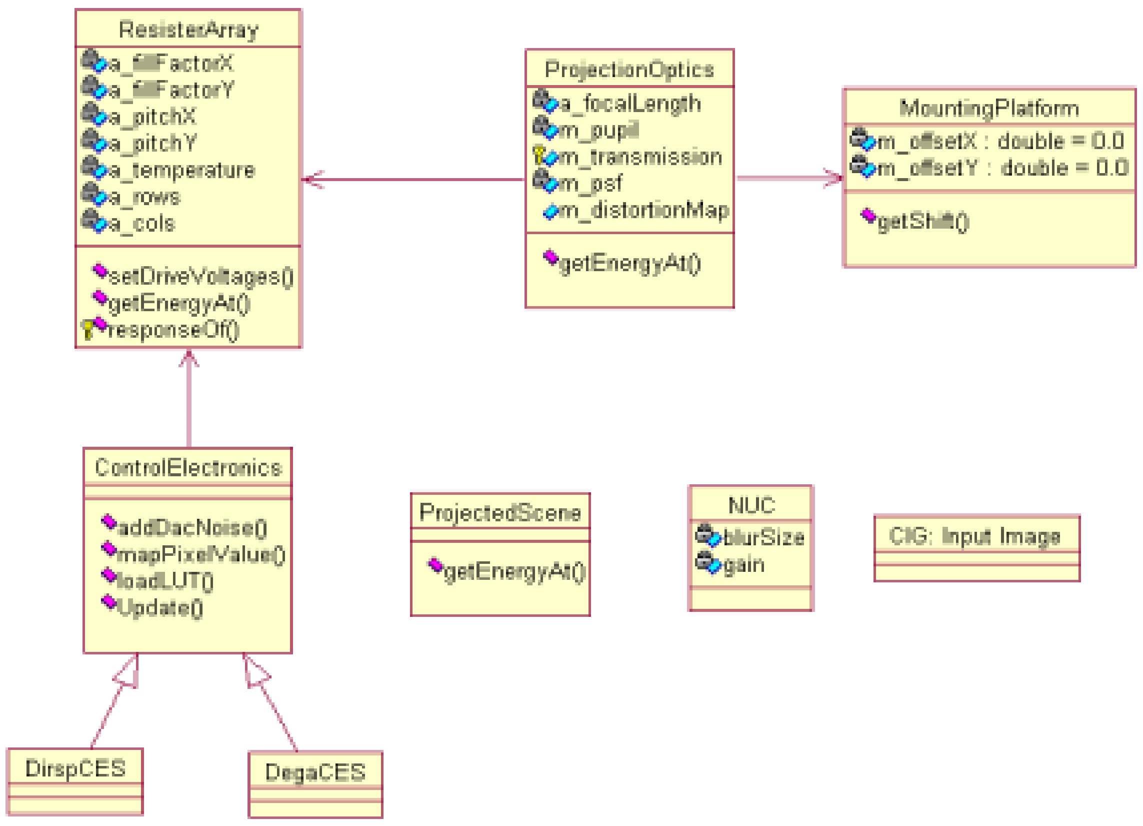

Figure 3 - Class Diagram

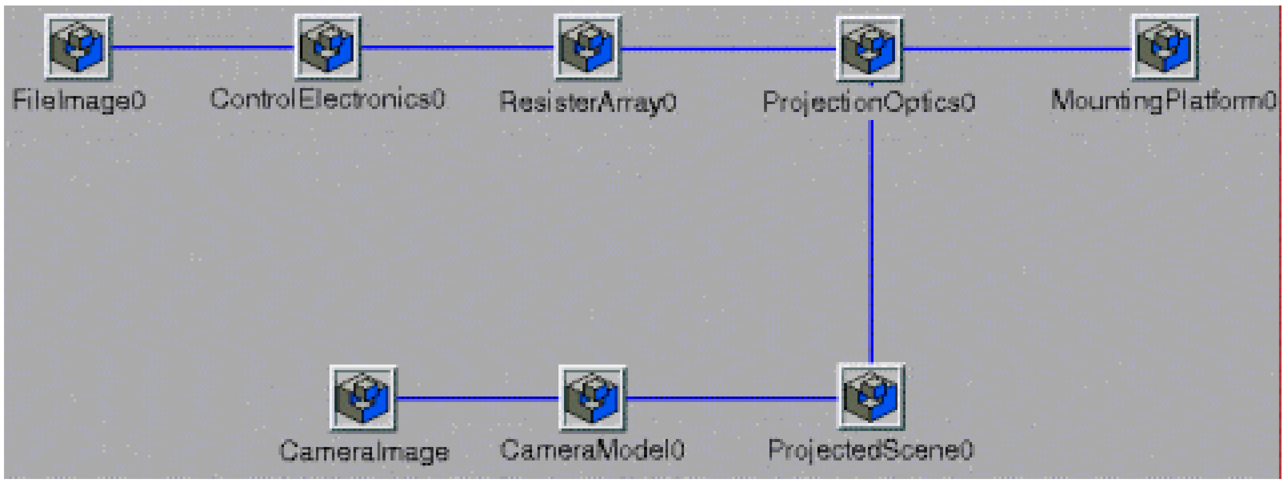

Figure 4 - Flow Chart

These illustrations give the developer the ability to “see” how the parts of the system will work. Using these diagrams the developer can design the blueprint of the software. Like in architecture, house blue prints will be modified numerous times before the house is actually built. It is much easier and less time consuming to modify the design before proceeding to implementation.

1.2.3 Implementation

The implementation of the IDM utilizes the advanced features of the C++ programming language and the Standard Template Library (STL) such as shared memory, maps, auto pointers, threads for parallel execution, templates, and many others. These features allow for the construction of a robust, object-oriented, and highly extensible simulation framework.

Each hardware component shown in the class diagrams was implemented as a class with its specific data and functionality encapsulated. The class diagram, Figure 3, displays each class, its attributes, and functions. The classes communicate with each other through shared memory. By way of encapsulation, the communication interfaces remain static even when the internal data of a class is changed. This prevents one class from "breaking" the whole system.

The parallel execution portions of the software are provided by a C++ template function called PapplyWork. This function handles the creation and spawning of threads. In addition to this, the function uses a "work pile" algorithm in which threads compete for chunks of work to achieve load balancing. Taking advantage of the multiple processors available increases the efficiency of the IDM simulation.

Following the CMM guidelines, all the IDM software is maintained under source code control using a product called SourceSafe, by Microsoft. This prevents more than one programmer from writing to a file at a time. A programmer must check-out a file to make changes, test his changes, and then check the file back into the database for the other programmers to retrieve. This ensures the software integrity and maintains records of all changes made to the software.

1.2.4 Testing

The testing phase of the IDM is currently being performed in-house and by an external beta reviewer. Following the CMM, a Test Plan document is written with a specific test for each requirement and step-by-step instructions on how the user is to perform each test. The beta tester is then to write up a test report, citing the findings.

The requirements matrix is used to trace through the software to ensure that each requirement is met in the code. Together the requirements and test procedures are then put into what is called a Traceability Matrix that maps the requirement identifiers to the Validation Test Cases written in the Test Plan.

Unit testing is a process that uses individual tests for every class and its functions. By using macros, expressions are tested and errors are automatically reported in a customizable format. In the IDM, we refer to this as the C++ Testing Framework (CTF). In addition to the formal testing, this form of testing is being performed on the foundation classes utilized in the IDM.

Test Case	Requirement ID	Requirement
1	1.5.2.1	Must represent the basic IRSP hardware components: resister Array, Camera, Projection Optics, Control Electronics.
2	1.5.2.2	Scaled speedup exceeds 40% of optimum on 8 processors or greater. Definition: Optimum Theory is defined as 1 x the work can be done on 1 processor then optimally 8X the work should be able to be done on 8 process.
3	1.5.2.3	Parallel speedup reduces clock time by 2 times on an 8 processor system compared to a single processor system
4	1.5.2.4	IRSP digital model (IDM) & Scene-based Infrared Scene Generator (IRSG) NUC (SBNUC)Codes will run on two HPC platforms with same valid results
5	1.5.2.5	All documents and source code under configuration management
6	1.5.2.6	Option to output derived pixel data in a window

Figure 5 - Traceability Matrix

1.3 Conclusions

Following the guidelines of the CMM and the basic OOM have made the IDM software development process run smoothly. The CMM advises keeping accurate and concise records of the financials and the software development process. By doing this on the IDM project, we have a history of all the financials and the technical flow of the software development. When necessary, we are able to go back to the documentation for reference. The software engineering protocols of the CMM from requirements to testing have allowed the developers to thoroughly evaluate and implement the IDM system.

Object oriented methods provide an environment for building a flexible, reusable, and extensible system. The IDM software design is built on the concept of objects, this will allow for future growth. Through the use of encapsulation and inheritance, it would be simple to add new hardware components or modify existing ones to the IDM. The extensibility of the IDM provides a future open to many possibilities.

1.4 References

1. Bush, Marylin; Chrissis, Mary Beth et al: “Key Practices of the Capability Maturity Model”, Technical Report, Carnegie Mellon University – Software Engineering Institute, February 1993.

2. Pressman, Roger S. Ph.D.: “Software Engineering: A Practitioner’s Approach” Fourth Edition, The McGraw-Hill Companies, Inc. 1997.

3. Schildt, Herbert: “Teach Yourself C++”, Second Edition, Osborne McGraw-Hill, 1994.

4. Spears, J. Brent and Gossage, Brett N.: “ An Object-Oriented Software Framework for Execution of Real-Time, Parallel Algorithms”, 2001 International Conference on Computational Science, May 28.

5. Stroustrup, Bjarne:. “The C++ Programming Language”, Third Edition, AT&T Labs, 1997.